Continuous Timestamping for Efficient Replication Management in DHTs

Reza Akbarinia, Mounir Tlili, Esther Pacitti, Patrick Valduriez, Alexandre A. B. Lima

Continuous Timestamping for Efficient Replication Management in DHTs

Reza Akbarinia[1], Mounir Tlili[2], Esther Pacitti[3], Patrick Valduriez[4], Alexandre A. B. Lima[5]

[1,2]INRIA and LINA, Univ. Nantes, France
[3]LIRMM and INRIA, Univ. Montpellier, France
[4]INRIA and LIRMM, Montpellier, France
[5]COPPE/UFRJ, Rio de Janeiro, Brazil
[1,4]Firstname.Lastname@inria.fr, [2]pacitti@lirmm.fr, [3]Firstname.Lastname@univ-nantes.fr, [5]assis@cos.ufrj.br

Abstract. Distributed Hash Tables (DHTs) provide an efficient solution for data location and lookup in large-scale P2P systems. However, it is up to the applications to deal with the availability of the data they store in the DHT, e.g. via replication. To improve data availability, most DHT applications rely on data replication. However, efficient replication management is quite challenging, in particular because of concurrent and missed updates. In this paper, we propose an efficient solution to data replication in DHTs. We propose a new service, called Continuous Timestamp based Replication Management (CTRM), which deals with the efficient storage, retrieval and updating of replicas in DHTs. To perform updates on replicas, we propose a new protocol that stamps update actions with timestamps generated in a distributed fashion. Timestamps are not only monotonically increasing but also continuous, i.e. without gap. The property of monotonically increasing allows applications to determine a total order on updates. The other property, i.e. continuity, enables applications to deal with missed updates. We evaluated the performance of our solution through simulation and experimentation. The results show its effectiveness for replication management in DHTs.

1   Introduction

Distributed Hash Tables (DHTs), *e.g.* CAN [7] and Chord [10], provide an efficient solution for data location and lookup in large-scale P2P systems. While there are significant implementation differences between DHTs, they all map a given key k onto a peer p using a hash function and can lookup p efficiently, usually in O(*log n)* routing hops, where n is the number of peers [2]. One of the main characteristics of DHTs (and other P2P systems) is the dynamic behavior of peers which can join and leave the system frequently, at any time. When a peer gets offline, its data becomes unavailable. To improve data availability, most applications which are built on top of DHTs, rely on data replication by storing the (*key, data*) pairs at several peers, *e.g.* using several hash functions. If one peer is unavailable, its data can still be retrieved from the other peers that hold a replica. However, update management is difficult

because of the dynamic behaviour of peers and concurrent updates. There may be *replica holders* (i.e. peers that maintain replicas) that do not receive the updates, e.g. because they are absent during the update operation. Thus, we need a mechanism that efficiently determines whether a replica on a peer is up-to-date, despite missed updates. In addition, to deal with concurrent updates, we need to determine a total order on the update operations.

In this paper, we give an efficient solution to replication management in DHTs. We propose a new service, called Continuous Timestamp based Replication Management (CTRM), which deals with the efficient storage, retrieval and updating of replicas in DHTs. To perform updates on replicas, we propose a new protocol that stamps the updates with timestamps which are generated in a distributed fashion using groups of peers managed dynamically. The updates' timestamps are not only monotonically increasing but also continuous, i.e. without gap. The property of monotonically increasing allows CTRM to determine a total order on updates and to deal with concurrent updates. The continuity of timestamps enables replica holders to detect the existence of missed updates by looking at the timestamps of the updates they have received. Examples of applications that can take advantage of continuous timestamping are the P2P collaborative text editing applications, e.g. P2P Wiki [11], which need to reconcile the updates done by collaborating users. We evaluated our CTRM service through experimentation and simulation; the results show its effectiveness. In our experiments, we compared CTRM with two baseline services, and the results show that with a low overhead in update response time, CTRM supports fault tolerant data replication using continuous timestamps. The results also show that data retrieval with CTRM is much more efficient than the baseline services. We investigated the effect of peer failures on the correctness of CTRM, the results show that it works correctly even in the presence of peer failures.

The rest of this paper is organized as follows. In Section 2, we define the problem we address in this paper. In Section 3, we propose our replication management service, CTRM. Section 4 describes a performance evaluation of our solution. Section 5 discusses related work. Section 6 concludes.

2   Problem Definition

In this paper we deal with improving data availability in DHTs. Like several other protocols and applications designed over DHTs, e.g. [2], we assume that the lookup service of the DHT behaves properly. That is, given a key *k*, it either finds correctly the responsible for *k* or reports an error, *e.g.* in the case of network partitioning where the responsible peer is not reachable.

To improve data availability, we replicate each *object* (for instance a file) at a group of peers of the DHT which will call *replica holders*. Each replica holder keeps a *replica copy* of a DHT object. Each replica may be updated locally by a replica holder or remotely by other peers of the DHT. This model is in conformance with the *multi-master replication* model [5]. Updates on replicas are asynchronous, i.e., an update is applied first to a replica and afterwards (after the update's commitment), to the other replicas of the same object.

The problem that arises is that a replica holder may fail or leave the system at any time. Then, when re-joining (or recovering) it may need to retrieve the updates it missed when gone. Furthermore, updates on different replica copies of an object may be performed in parallel. To ensure consistency, updates must be applied to all replicas in a specific total order.

In this model, to ensure consistency of replicas, we need a distributed mechanism that determines 1) a total order for the updates; 2) the number of missed updates at a replica holder. Such a mechanism allows dealing with concurrent updates, i.e. committing them in the same order at all replica holders. In addition, it allows a rejoining (recovering) replica holder to determine whether its local replica is up-to-date or not, and how many updates should be applied on the replica if it is not up-to-date.

One solution for realizing such a mechanism is to stamp the updates with timestamps that are monotonically increasing and continuous. We call such a mechanism *update with continuous timestamps*.

Let *patch* be the action (or set of actions) generated by a peer during one update operation. Then, the property of update with continuous timestamps can be defined as follows.

Definition 1: Update with continuous timestamps (UCT). A mechanism of update is UCT *iff* patches of updates are stamped by increasing real numbers such that, for any two consecutive committed updates, the difference between their timestamps is one.

Formally, consider two consecutive committed updates u_1 and u_2 on a data d, and let pch_1 and pch_2 be the patches of u_1 and u_2, respectively. Assume that u_2 is done after u_1, and let t_1 and t_2 be the timestamps of pch_1 and pch_2 respectively. Then we should have $t_2 = t_1 + 1$;

To support the UCT property in a DHT, we must deal with two challenges: 1) To generate continuous timestamps in the DHT in a distributed fashion; 2) To ensure that any two consecutive generated timestamps are used for two consecutive updates. Dealing with the first challenge is hard, in particular due to the dynamic behavior of peers which can leave or join the system at any time and frequently. This behavior makes inappropriate the timestamping solutions based on physical clocks, because the distributed clock synchronization algorithms do not guarantee good synchronization precision if the nodes are not linked together long enough [6]. Addressing the second challenge is difficult as well, because there may be generated timestamps which are used for no update, e.g. because the timestamp requester peer may fail before doing the update.

3 Replication Management

In this section, we propose a replication management service, called Continuous Timestamp based Replication Management (CTRM), that deals with efficient storage, retrieval and updating of replicas on top of DHTs, while supporting the UCT property. The rest of this section is organized as follows. We firstly give an overview of CTRM. Secondly, we introduce the concept of replica holder groups which is an

efficient approach for replica storage by CTRM. Thirdly, we propose a new protocol used by CTRM for performing updates on replicas. Finally, we show how CTRM deals with peer faults that may happen during the execution of the protocol.

3.1 Overview

To provide high data availability, CTRM replicates each data in the DHT at a group of peers, called *replica holder group*, determined by using a hash function. After each update on a data, the corresponding patch is sent to the group where a monotonically increasing timestamp is generated by one of the members, i.e. the responsible of the group. Then the patch and its timestamp are published to the members of the group using an update protocol, called UCT protocol (see the details in Section 3.3).

To retrieve an up-to-date replica of a data, the request is sent to the responsible of the data's replica holder group. The responsible peer sends the data and the latest generated timestamp to the group members, one by one, and the first member that has received all patches returns its replica to the requester. To verify whether all patches are received, replica holders check the two following conditions, called *up-to-date conditions*: 1) the timestamps of the received patches are continuous, i.e. there is no missed update; 2) the latest generated timestamp is equal to the timestamp of the latest patch received by the replica holder.

The above up-to-date conditions are also verified periodically by each member of the group. If the conditions do not hold, the member updates its replica by retrieving the missed patches and their corresponding timestamps from the responsible of the group or other members that hold them.

3.2 Replica Holder Groups

Let G_k be the group of peers that maintain the replicas of a data whose ID is k. We call these peers the *replica holders for k*. For replica holders of each data, we use peers that are relatively close in the overlay network. For each group, there is a responsible peer which is also one of its members. For choosing the responsible of the group G_k, we use a hash function h_r, and the peer p that is responsible for key=$h_r(k)$ in the DHT, is the responsible of G_k. In this paper, the peer that is responsible for key=$h_r(k)$ is denoted by *rsp(k , h_r), i.e.* called responsible of k with respect to hash function h_r. In addition to *rsp(k , h_r)*, some of the peers that are close to it, .e.g. its neighbors, are members of G_k. Each member of the group knows the address of other members of the group. The number of members of a replica holders group, i.e. $\left| G_k \right|$, is a system's parameter.

If the peer p that is responsible for a group leaves the system or fails, another peer, say q, becomes responsible for the group, i.e. the new responsible of the key=$h_r(k)$ in the DHT. In almost all DHTs (e.g. CAN [7] and Chord [10]), the new responsible peer is one of the neighbors of the previous one.

<div style="border:1px solid">

1. On update requester:
   - Send *{k, pch}* to *rsp(k, h_r)*
   - Monitor *rsp(k, h_r)* using a failure detector
   - Go to Step 8 if *rsp(k, h_r)* fails

2. On *rsp(k, h_r)*: upon receiving *{k, pch}*
   - Set $c_k = c_k + 1$; // increase counter by one
     // initially we have $c_k = 0$;
   - Let $ts = c_k$, send *{k, pch, ts}* to other replica holders;
   - Set a timer on, called ackTimer, to a default time

3. On each replica holder: upon receiving *{k, pch, ts}*
   - Maintain *{k, pch, ts}* in a temporary memory on disk;
   - Send ack to *rsp(k, h_r)*;

4. On *rsp(k, h_r)*: upon expiring ackTimer
   - If (number of received acks $\geq$ threshold δ) then send "commit" message to the replica holders;
   - Else set $c_k = c_k - 1$, and send "abort" message to the update requester;

5. On each replica holder: upon receiving "commit"
   - Maintain *{pch, ts}* as a committed patch for *k*.
   - Update the local replica using *pch*;
   - Send "terminate" message to *rsp(k, h_r)*

6. On *rsp(k, h_r)*: upon receiving the first 'terminate' message
   - Send "terminate" to update requester

7. On update requester: receiving the 'terminate' from *rsp(k, h_r)*
   - Commit the update operation

8. On update requester: upon detecting a failure on *rsp(k, h_r)*
   - If the 'terminate' message is received then commit the update operation;
   - Else, check replica holders, if at least one of them received the 'commit' message then commit the update operation;
   - Else, abort the update operation;

</div>

Figure 1. UCT protocol

If a responsible peer *p* leaves the system normally, i.e. without fail, it sends to the next responsible peer, i.e. *q*, the last timestamps of all data replicated in the group. If *p* fails, then the next responsible peer, say *q*, contacts the members of the group (most of which are its neighbors) and asks them to return the timestamps which they maintain for the data replicated over them. Then, for each replicated data, *q* initializes a timestamp equal to the highest received timestamp from the group members.

Each group member *p* periodically sends alive messages to the responsible of the group, and the responsible peer returns to it the current list of members. If the responsible peer does not receive an alive message from a member, it assumes that the member has failed. When a member of a group leaves the system or fails, after getting aware of this departure, the responsible of the group invites a close peer to join the group, *e.g.* one of its neighbors. The new member receives from the responsible peer a list of other members as well as up-to-date replicas of all data replicated by the group.

Each peer can belong to several groups, but it can be responsible for only one group. Each group can hold the replicas of several data items.

3.3 Update with Continuous Timestamps

In this section, we propose a protocol, called UCT (Update with Continuous Timestamps) that deals with updating replicas in CTRM.

To simplify the description of our update protocol, we assume the existence of (not perfect) failure detectors [3] that can be implemented as follows. When we setup a

failure detector on a peer p to monitor peer q, the failure detector periodically sends ping messages to q in order to test whether q is still alive (and connected). If the failure detector receives no response from q, then it considers q as a failed peer, and triggers an error message to inform p about this failure.

Let us now describe the UCT protocol. Let p_0 be the peer that wants to update a data whose ID is k. The peer p_0 is called update requester. Let pch be the patch of the update performed by p_0. Let p_1 be the responsible of the replica holder group for k, *i.e.* $p_1 = rsp(k, h_r)$. The protocol proceeds as follows (see Figure 1):

- **Update request.** In this phase, the update requester, *i.e.* p_0, obtains the address of the responsible of the replica holder group, *i.e.* p_1, by using the DHT's lookup service, and sends to it an update request containing the pair *(k, pch)*. Then, p_0 waits for a commit message from p_1. It also uses a failure detector and monitors p_1. The wait time is limited by a default value, *e.g.* by using a timer. If p_0 receives the terminate message from p_1, then it commits the operation. If the timer timeouts or the failure detector reports a fault of p_1, then p_0 checks whether the update has been done or not, *i.e.* by checking the data at replica holders. If the answer is positive, then the operation is committed, else it is aborted.

- **Timestamp generation and replica publication.** After receiving the update request, p_1 generates a timestamp for k, *e.g.* ts, by increasing a local counter that it keeps for k, say c_k. Then, it sends *(k, pch, ts)* to the replica holders, *i.e.* the members of its group, and asks them to return an acknowledgement. When a replica holder receives *(k, pch, ts)*, it returns the acknowledgement to p_1 and maintains the data in a temporary memory on disk. The patch is not considered as an update before receiving a commit message from p_1. If the number of received acknowledgements is more than or equal to a threshold δ, then p_1 starts the update confirmation phase. Otherwise p_1 sends an abort message to p_0. The threshold δ is a system parameter, e.g. it is chosen in such a way that the probability that δ peers of the group simultaneously fail is almost zero.

- **Update confirmation.** In this phase, p_1 sends the commit message to the replica holders. When a replica holder receives the commit message, it labels *{pch, ts}* as a committed patch for k. Then, it executes the patch on its local replica, and sends a terminate message to p_1. After receiving the first terminate message from replica holders, p_1 sends a terminate message to p_0. If a replica holder does not receive the commit message for a patch, it discards the patch upon receiving a new patch containing the same or greater timestamp value.

Notice that the goal of our protocol is not to provide eager replication, but to have at least δ replica holders that receive the patch and its timestamp. If this goal is attained, the update operation is committed. Otherwise it is aborted, and the update requester should try its update later.

Let us now consider the case of concurrent updates, e.g. two or more peers want to update a data d at the same time. In this case, the concurrent peers send their request to the responsible of the d's group, say p_1. The peer p_1 determines an order for the requests, e.g. depending on their arrival time or on the distance of requesters if the

requests arrive at the same time. Then it processes the requests one by one according their order, i.e. it commits or aborts one request and starts the next one. Thus, concurrent updates make no problem of inconsistency for our replication management service.

3.4 Fault Tolerance

Let us now study the effect of peer failures on the UCT protocol and discuss how they are handled. By peer failures, we mean the situations where a peer crashes or gets disconnected from the network abnormally, e.g. without informing the responsible of the group. We show that these failures do not block our update protocol. We also show that even in the presence of these failures, the protocol guarantees continuous timestamping, *i.e.* when an update is committed, the timestamp of its patch is only one unit greater than that of the previous one. For this, it is sufficient to show that each generated timestamp is attached with a committed patch, or it is aborted. By aborting a timestamp, we mean returning the counter's value to its value before the update operation.

During our update protocol, a failure may happen on the responsible of the group or on a replica holder. We first study the case of the responsible of the group. In this case, the failure may happen in one of the following time intervals:

- *I_1*: **after receiving the update request and before generating the timestamp.** If the responsible of the group fails in this interval, then after some time, the failure detector detects the failure or the timer timeouts. Afterwards, the update requester checks the update at replica holders, and since it has not been done, the operation is aborted. Therefore, a failure in this interval does not block the protocol, and continuous timestamping is assured, *i.e.* because no update is performed.

- *I_2*: **after I_1 and before sending the patch to replica holders.** In this interval, like in the previous one, the failure detector detects the failure or the timer timeouts, and thus the operation is aborted. The timestamp *ts*, which is generated by the failed responsible peer, is aborted as follows. When the responsible peer fails, its counters get invalid, and the next responsible peer initializes its counter using the greatest timestamp of the committed patches at replica holders. Thus, the counter returns to its value before the update operation. Therefore, in the case of crash in this interval, continuous timestamping is assured.

- *I_3*: **after I_2 and before sending the commit message to replica holders.** If the responsible peer fails in this interval, since the replica holders have not received the commit, they do not consider their received data as a valid replica. Thus, when the update requester checks the update, they answer that the update has not been done and the operation gets aborted. Therefore, in this case, continuous timestamping is not violated.

- **I_4: after I_3 and before sending the terminate message to the update requester.** In this case, after detecting the failure or timeout, the update requester checks the status of the update in the DHT and finds out that the update has been done, thus it commits the operation. In this case, the update is done with a timestamp which is one unit greater than that of the previous update, thus the property of continuous timestamping is enforced.

4   Experimental Validation

In this section, we evaluate the performance of CTRM through experimentation over a 64-node cluster and simulation. The experimentation over the cluster was useful to validate our algorithm and calibrate our simulator. The simulation allows us to study scale up to high numbers of peers (up to 10,000 peers).

4.1   Experimental and Simulation Setup

Our experimentation is based on an implementation of the Chord [10] protocol. We tested our algorithms over a cluster of 64 nodes connected by a 1-Gbps network. Each node has two Intel Xeon 2.4 GHz processors, and runs the Linux operating system. To study the scalability of CTRM far beyond 64 peers, we also implemented a simulator using SimJava. After calibration of the simulator, we obtained simulation results similar to the implementation results up to 64 peers.

Our default settings for different experimental parameters are as follows. The latency between any two peers is a random number with normal distribution and a mean of 100 ms. The bandwidth between peers is also a random number with normal distribution and a mean of 56 Kbps (as in [1]). The simulator allows us to perform tests with up to 10,000 peers, after which simulation data no longer fit in RAM and makes our tests difficult. Therefore, the default number of peers is set to 10,000.

In our experiments, we consider a dynamic P2P system, *i.e.* there are peers that leave or join the system. Peer departures are timed by a random Poisson process (as in [8]). The average rate, *i.e.* λ, for events of the Poisson process is $\lambda=1$/second. At each event, we select a peer to depart uniformly at random. Each time a peer goes away, another joins, thus keeping the total number of peers constant (as in [8]).

We also consider peer failures. Let *fail rate* be a parameter that denotes the percentage of peers that leave the system due to a fail. When a peer departure event occurs, our simulator should decide on the type of this departure, i.e. normal leave or fail. For this, it generates a random number which is uniformly distributed in [0..100]; if the number is greater than *fail rate* then the peer departure is considered as a normal leave, else as a fail. In our tests, the default setting for *fail rate* is 5% (as in [1]). In our tests, unless otherwise specified, the number of replicas of each data is 10.

Although they cannot provide the same functionality as CTRM, the closest prior works to CTRM are the BRICKS project [4], denoted as BRK, and the Update Management Service (UMS) [1]. The assumptions made by these two works are close to ours, e.g. they do not assume the existence of powerful servers. BRK stores the

data in the DHT using multiple keys, which are correlated to the data key. To be able to retrieve an up-to-date replica, BRK uses versioning, i.e. each replica has a version number which is increased after each update. UMS has been proposed to support data currency in DHTs, i.e. the ability to return an up-to-date replica. It uses a set of m hash functions and replicates the data randomly at m different peers. UMS works based on timestamping, but the generated timestamps are not necessarily continuous.

4.2   Update Cost

Let us first investigate the performance of CTRM's update protocol. We measure the performance of data update in terms of response time and communication cost. By update response time, we mean the time needed to send the patch of an update operation to the peers that maintain the replicas. By update communication cost, we mean the number of messages needed to update a data.

Using our simulator, we ran experiments to study how the response time increases with the addition of peers. Using the simulator, Figure 2 depicts the total number of messages while increasing the number of peers up to 10,000, with the other simulation parameters set as defaults described in Section 4.1. In all three services, the communication cost increases logarithmically with the number of peers. However, the communication cost of CTRM is much better than that of UMS and BRK. The reason is that UMS and BRK perform multiple lookups in the DHT, but CTRM does only one lookup, i.e. only for finding the responsible of the group. Notice that each lookup needs $O(log\ n)$ messages where n is the number of peers of the DHT.

Figure 3 shows the update response time with the addition of peers up to 10,000, with the other parameters set as described in Section 4.1. The response time of CTRM is a little bit higher than that of UMS and BRK. The reason is that for guaranteeing continuous timestamping, the update protocol of CTRM performs two round-tips between the responsible of the group and the other members of the group. But, UMS and BRK only send the update actions to the replica holders by looking up the replica holders in parallel (note that the impact of parallel lookups on response time is very slight, but they have a high impact on communication cost). However, the difference in the response time of CTRM and that of UMS and BRK is small because the round-trips in the group are less time consuming than lookups. This slight increase in response time of CTRM's update operation is the price to pay for guaranteeing continuous timestamping.

4.3   Data Retrieval Response Time

We now investigate the data retrieval response time of CTRM. By data retrieval response time, we mean the time to return an up-to-date replica to the user.

Figure 4 shows the response time of CTRM, UMS and BRK with the addition of peers up to 10000, with the other parameters set as defaults described in Section 4.1. The response time of CTRM is much better than that of UMS and BRK. This difference in response time can be explained as follows. Both CTRM and UMS services contact some replica holders, say r, in order to find an up-to-date replica, e.g.
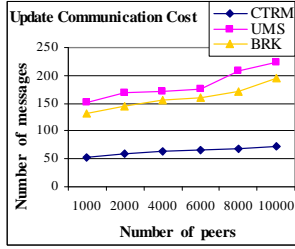
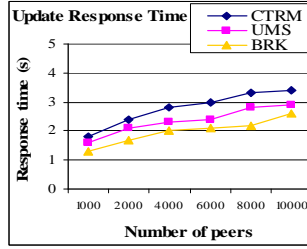**Figure 2.** Communication cost of updates vs. number of peers

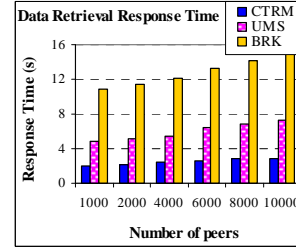**Figure 3.** Response time of update operation vs. number of peers

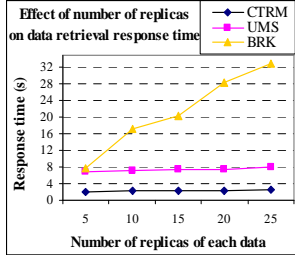**Figure 4.** Response time of data retrievals vs. number of peers

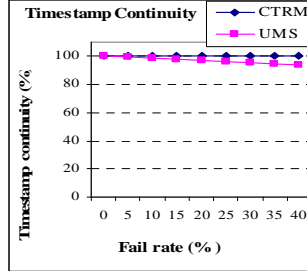**Figure 5.** Effect of the number of replicas on response time of data retrievals
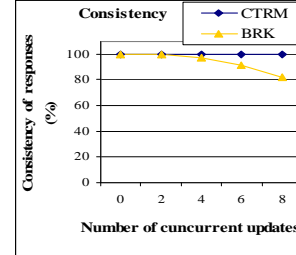

Figure 6. Timestamp continuity vs. fail rate

Figure 7. Consistency of returned results vs. number of concurrent updates

r=6. For contacting these replica holders, CTRM performs only one lookup (to find the responsible of the group) and some low-cost communications in the group. But, UMS performs exactly *r* lookups in the DHT. BRK retrieves all replicas of data from the DHT (to determine the latest version), and for each replica it performs one lookup. Thus the number of lookups done by BRK is equal to the total number of data replicas, i.e. 10 in our experiments.

Let us now study the effect of the number of replicas of each data, say *m*, on performance of data retrieval. Figure 5 shows the response time of data retrieval for the three solutions with varying the number of replicas up to 30. The number of replicas has almost a linear impact on the response time of BRK, because to retrieve an up-to-date replica it has to retrieve all replicas by doing one lookup for each replica. But, it has a slight impact on CTRM, because for finding an up-to-date replica CTRM performs only one lookup, and some low cost communications, i.e. in the group.

4.4 Effect of Peer Failures on Timestamps Continuity

Let us now study the effect of peer failures on the continuity of timestamps used for data updates. This study is done only for CTRM and UMS that work based on timestamping. In our experiments we measure *timestamp continuity rate* by which we mean the percentage of the updates whose timestamps are only one unit higher than that of their precedent update. We varied the fail rate parameter, and observed its effect on timestamp continuity rate.

Figure 6 shows timestamp continuity rate for CTRM and UMS while increasing the fail rate, with the other parameters set as described in Section 4.1. The peer failures do not have any negative impact on the continuity of timestamps generated by CTRM, because our protocol assures timestamp continuity. However, when

increasing the fail rate in UMS, the percentage of updates whose timestamps are not continuous increases.

4.5   Effect of Concurrent Updates on Result Consistency

In this section, we investigate the effect of concurrent updates on the consistency of the results returned by CTRM. In our experiments, we perform u updates done concurrently by u different peers using the CTRM service, and after finishing the concurrent updates, we invoke the service's data retrieval operation from n randomly chosen peers (n=50 in our experiments). If there is any difference between the data returned to the n peers, we consider the result as inconsistent. We repeat each experiment several times, and report the percentage of the experiments where the results are consistent. We perform the same experiments using the BRK service.

Figure 7 shows the results with the number of concurrent updates, i.e. u, increasing up to 8, and with the other parameters set as defaults described in Section 4.1. As shown, in 100% of experiments the results returned by CTRM are consistent. This shows that our update protocol works correctly even in the presence of concurrent updates. However, the BRK service cannot guarantee the consistency of results in the case of concurrent updates, because two different updates may have the same version at different replica holders.

5   Related Work

Most existing P2P systems support data replication, but usually they do not deal with concurrent and missed updates.

OceanStore [9] is a data management system designed to provide a highly available storage utility on top of P2P systems. It allows concurrent updates on replicated data, and relies on reconciliation to assure data consistency. The reconciliation is done by a set of powerful servers using a consensus algorithm. The servers agree on which operations to apply, and in what order. However, in the applications, which we address, the presence of powerful servers is not guaranteed.

The BRICKS project [4] provides high data availability in DHTs through replication. For replicating a data, BRICKS stores the data in the DHT using multiple keys, which are correlated to the data key, e.g. k. There is a function that given k, determines its correlated keys. To be able to retrieve an up-to-date replica, BRICKS uses versioning. Each replica has a version number which is increased after each update. However, because of concurrent updates, it may happen that two different replicas have the same version number, thus making it impossible to decide which one is the latest replica.

In [1], an update management service, called UMS, was proposed to support data currency in DHTs, *i.e.* the ability to return an up-to-date replica. However, UMS does not guarantee continuous timestamping which is a main requirement for collaborative applications which need to reconcile replica updates. UMS uses a set of m hash functions and replicates randomly the data at m different peers, and this is more

expensive than the groups which we use in CTRM, particularly in terms of communication cost. A prototype based on UMS was demonstrated in [12].

6 Conclusion

In this paper, we addressed the problem of efficient replication management in DHTs. We proposed a new service, called continuous timestamp based replication management (CTRM), which deals with efficient and fault tolerant data replication, retrieval and update in DHTS, by taking advantage of replica holder groups and monotonically increasing and continuous timestamps.

References

[1]   Akbarinia, R., Pacitti, E., Valduriez, P.: Data Currency in Replicated DHTs. *SIGMOD Conf.*, 211-222 (2007)

[2]   Chawathe, Y., Ramabhadran, S., Ratnasamy, S., LaMarca, A., Shenker, S., Hellerstein, J.M.: A case study in building layered DHT applications. *SIGCOMM Conf.*, 97-108 (2005)

[3]   Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-Area Cooperative Storage with CFS. *ACM Symp. on Operating Systems Principles*, 202-215 (2001)

[4]   Knezevic, P., Wombacher, A., Risse, T.: Enabling High Data Availability in a DHT. *Proc. of Int. Workshop on Grid and P2P Computing Impacts on Large Scale Heterogeneous Distributed Database Systems*, 363-367 (2005)

[5]   Özsu, T., Valduriez, P.: *Principles of Distributed Database Systems*. 2nd Edition, Prentice Hall, 1999.

[6]   PalChaudhuri, S., Saha, A.K., Johnson, D.B.: Adaptive Clock Synchronization in Sensor Networks. *Int. Symp. on Information Processing in Sensor Networks*, 340-348 (2004)

[7]   Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. *SIGCOMM Conf.*, 161-172 (2001)

[8]   Rhea, S.C., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. *USENIX Annual Technical Conf.*, 127-140 (2004)

[9]   Rhea, S.C., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., Kubiatowicz, J.: Pond: the OceanStore Prototype. *USENIX Conf. on File and Storage Technologies*, 1-14 (2003)

[10]   Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F. Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. *SIGCOMM Conf.*, 149-160 (2001)

[11]   Xwiki Concerto Project: http://concerto.xwiki.com

[12]   Tlili, M., Dedzoe, W.K., Pacitti, E., Valduriez, P., Akbarinia, R., Molli, P., Canals, G., Laurière, S.: P2P logging and timestamping for reconciliation. *PVLDB 1(2)*: 1420-1423 (2008)